Accurately and Efficiently Estimating Dynamic Point-to-Point Shortest Path

Alok Tripathy*
atripathy8@gatech.edu

ABSTRACT

Point-to-point shortest path (PPSP), or *s-t* connectivity, is a variant of the shortest path problem found in graph theory. In this problem, we are given a graph and pairs of vertices over time, and the output is the shortest path between each pair of vertices. In this paper, we present two algorithms. Our first algorithm approximately solves the PPSP problem on any arbitrary graph. For each pair of vertices queried, it accurately and efficiently estimates the shortest path between the two vertices. Our second algorithm extends the first to work on dynamic graphs. That is, our second algorithm can efficiently account for changes in the graph, such as friend requests on the Facebook network or road closures on road networks. At a high level, both algorithms partition the graph into highly connected communities. To respond to a query $q(u, v)$, they each find the fewest number of partitions between u and v, and the shortest path through each partition. We show that our static graph algorithm can approximate the distance between two vertices with about $20\% - 35\%$ percent error and anywhere from $80X - 70000X$ faster than a BFS in practice with the right choice of partitions. Additionally, we show that our dynamic graph algorithm can account for updates to the graph anywhere from $20X - 20000X$ faster than rerunning the static graph algorithm for each change to the graph.

KEYWORDS

Graph analytics, Shortest paths, Point-to-point shortest path

ACM Reference Format:
Alok Tripathy. 2018. Accurately and Efficiently Estimating Dynamic Point-to-Point Shortest Path. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The applications for shortest paths in networks and graphs are widespread. They play a pivotal role in a wide variety of domains, such as biology, transportation, sociology, and engineering.

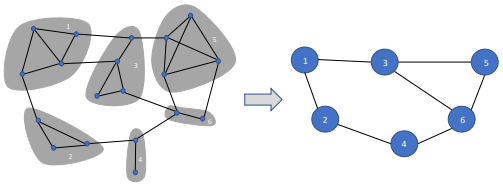


fig. 1: *Left: Input graph already partitioned. Right: Supergraph created from partitions.*

While Single-Source Shortest Path (SSSP) is a well-studied problem in computing, in practice, many SSSP methods and their variants do not perform well for a variety of reasons. For instance, computing the exact value of the shortest path does not scale well for large networks. All-pair shortest path (APSP) algorithms are also computationally expensive, and they do an excessive amount of work if the user only needs the shortest paths between a few vertex pairs. Thus, algorithms that compute shortest paths between vertex pairs of interest are desirable. This version of the shortest path problem is known as the *point-to-point shortest path problem* (PPSP). Alternatively, one can think of this problem as a multiple *s-t* connectivity instances.

Many PPSP and APSP algorithms, such as Goldberg *et. al.* [14] [13], target static graphs, i.e. graphs that do not change. In reality, networks change structure frequently, such as friend requests in a social network. Using static-graph algorithms, if an edge is either inserted or deleted from the graph, it is necessary to rerun the entire algorithm on the new graph. As more operations occur on the graph, this becomes extremely inefficient.

In this paper, we show a new and efficient algorithm for approximating distances between vertices for the PPSP problem on static graphs. Our algorithm does this by partitioning the input graph into smaller subgraphs. After partitioning, we create a new graph whose vertices represent subgraphs and whose edges are edges in the input graph that cross subgraphs. This new graph is known as the *supergraph*, as depicted in Figure 1. To approximate the shortest path between vertices u and v, our algorithm will find the fewest number of subgraphs between u and v, and then find the shortest path through each of these subgraphs. We will also extend our static graph algorithm to introduce an efficient dynamic graph algorithm, i.e. an algorithm that takes edge updates to the graph into account for future queries.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1.1 Summary of Results

Our main contributions in this paper consist of the following.

- We propose an algorithm that can efficiently approximate the shortest path between any two query vertices on static graphs. We show that this algorithm, with the right choice of partitions, approximates shortest paths usually with a 20%-35% error.
- We show that this algorithm algorithm can respond to shortest path queries in milliseconds, and $80X$-$70000X$ faster than a BFS.
- We extend this static-graph algorithm to develop a dynamic graph algorithm. That is, we extend our original algorithm to efficiently take edge insertions and deletions into account for future queries.

2 RELATED WORK

2.1 Problem Definition

In PPSP, we are given an input graph $G = (V, E)$ and are asked to respond to shortest path queries denoted as $q(u, v)$. A query $q(u, v)$ is a request by the user for the length of the shortest path from vertex u to vertex v. The number of such queries is unbounded, and can change overtime. The problem of PPSP and the more elaborate APSP can be expensive for large graphs—especially for dynamic ones where values constantly need to be updated. As such, we resort to computing approximate PPSP in the dynamic setting as well.

2.2 Distance Approximation

Shun [27] addresses the problem of computing eccentricities for static graphs. The eccentricity of a vertex u the largest value of $d(u, v)$ for all $v \in V$, where $d(u, v)$ is the length of the shortest path from u to v. Shun shows that the k-BFS algorithm is empirically more accurate and more efficient than theoretically more sophisticated algorithms.

Potamias et. al. [25], Goldberg *et. al* [14], and Goldberg [13] present several approximation algorithms for static graphs. Most use a **landmark-based** algorithm where the estimate for a query $d(u, v)$ is simply $d(u, l) + d(l, v)$ for some vertex l, with the distance from l to every other vertex determined in preprocessing. Their approach is very different than the one we take here and does not work for dynamic graphs.

Crescenzi *et. al.* [9] compare three randomized algorithms that estimate the distance distribution among vertices in a graph. Berman and Kasiviswanathan [4] propose a theoretical approximation algorithm for APSP on weighted graphs s.t. that each the distance between each vertex pair is bounded by the largest weight edge between them. Chechik *et. al.* [8] also present two $\frac{3}{2}$-approximation algorithms for computing a graph's diameter with two different run-times: $\widetilde{O}(E^{\frac{3}{2}})$ and $\widetilde{O}(E \cdot V^{\frac{2}{3}})$. This $\frac{3}{2}$-approximation means that Chechik's algorithms run with a theoretical bound of 50% error. Our algorithm, in practice, beats this number on multiple accounts with errors ranging from 10% to 20% while dealing with dynamic graphs and scaling to large graphs.

2.3 Dynamic APSP and BC

In order for our algorithm to solve dynamic PPSP, it is important to consider dynamic algorithms for similar problems. One such problem is dynamic APSP. Demetrescu and Italiano [7] present a non-approximate solution to the problem that takes $\widetilde{O}(V^2)$ amortized time for updates to the graph. King also presents a solution that has an amortized update cost of $O(V^2 \log^2 V / \log \log V)$ with error factor $(2 + \epsilon)$, $O(V^2 \log^3(bV)/\epsilon^2)$ with error factor $(1 + \epsilon)$, and $O(V^{2.5}\sqrt{b \log V})$ for an exact solution, where b is the largest weight edge in the graph [20]. Ramalingan and Reps also propose an algorithm for dynamic APSP [26]. Furthermore, Demetrescu, Emilozzi, and Italiano argue that all of these algorithms are efficient in practice [6].

In recent times, several practical implementations of dynamic APSP have been designed for betweenness centrality, which is a variant of APSP that focuses on finding key vertices in a graph based on the number of shortest paths through each vertex. APSP is the backbone of BC, so dynamic BC algorithms could help us as well. Fortunately, there exist several dynamic BC algorithms, such as those due to Lee *et al.* [21], Kas *et al.* [19], Bergamini *et al.* [3] (two algorithms), Nasre *et al.*. [24], and Green *et al.* [16]. Furthermore, all approximate algorithms can also be extended to return exact results but the other the way is not true.

Of these algorithms, the only ones that can be applied to both undirected and approximation algorithms are those by Bergamini *et al.* [3] and Green *et al.* [16]. We choose to use Green's algorithm due to its simplicity, accuracy, scalability, and performance in practice.

2.4 Dynamic Graph Data Structures

As our algorithm is designed for dynamic graphs, it is both desirable and necessary that a dynamic graph data structure be used that adds as little overhead as possible whenever the graph is updated. Many actual implementations rebuild the graph representation after each update—that is computationally expensive and undesirable. There are numerous such data structures in the literature, such as Pegasus [10], Giraph [17], GraphLab [23], *STINGER* [1, 11], cuSTINGER [15] and Hornet [5]. cuSTINGER and Hornet are recent dynamic graph data structures designed for the GPU. As our current algorithm is implemented for shared memory systems we focus on the *STINGER* data structure which was designed for shared-memory systems. STINGER can be thought of as a hybrid of the adjacency list and Compressed Sparse Row (CSR) representations of graphs. Every vertex v in the graph has a linked list of *edge blocks*, where an edge block is a small array of a subset of v's neighbors. This makes STINGER better suited for dynamic graphs than the sparse Compressed Sparse Row (CSR) data structure which is immutable. In our experiments we confirmed that updating STINGER was not an execution bottleneck and such will not reduce the performance of our new algorithm.

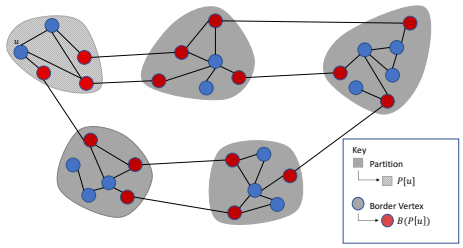


fig. 2: *Illustration of partitions and border vertices*

Table 1: *List of symbols and notations used by our algorithm*

Symbols and Notations	
Symbol	Description
G	Input graph.
V	Set of vertices in the input graph.
E	Set of edges in the input graph.
P	Set of all partitions of input graph.
$P[u]$	Partition of vertex u in G.
p	Partition in G.
bv	Individual border vertex (i.e. vertex with edge that crosses partitions).
$G^S = (V^S, E^S)$	Supergraph of G, V^S is the set of partitions in P and E^S is the set of edges in E the cross partitions.
$q(s,t)$	Query for the shortest path length between vertices u and v.
Q	Query list.
$B(p)$	Set of border vertices in partition p.
Notations and Fields	
$p.dist[u][v]$	Shortest path length from u to v within partition p.
$dist[u][v]$	Shortest path length from u to v in the supergraph.
$next[p]$	Supervertex after p on shortest path from $P[u]$ to $P[v]$ for query $q(u,v)$.

3 STATIC GRAPH ALGORITHM

3.1 Algorithm Description

Our algorithm consists of two main parts: the *preprocessing* phase and the *querying* phase. The latter is depicted in Algorithms 1 and 2. The goal of preprocessing is to partition the graph into a set of highly-connected partitions P and construct the supergraph $G^S = (V^S, E^S)$, where the supergraph is another representation of G. V^S are the partitions of G, and E^S are edges between two vertices in different partitions. We also define a special subset of vertices within any partition p known as the **border vertices** of p (denoted by $B(p)$). These are vertices in p that have an edge to a vertex in another partition, as depicted in Figure 2.

In the querying phase, we, naturally, begin to respond to queries. For each query $q(s,t)$, we use the partition $P[s]$ and $P[t]$ for the vertices s and t, respectively, and find the shortest path between $P[s]$ and $P[t]$ in the supergraph. Then, the algorithm finds the quickest path through each partition on this shortest path between $P[s]$ and $P[t]$ in the supergraph. In other words, for any query $q(s,t)$, we find the fewest number of partitions between s and t and the quickest way through each partition.

At a high level, the reason this algorithm runs efficiently in practice is that, no matter the size of the input graph, queries never have to deal with a large portion of the graph. If s and t lie in the same partition p, then the algorithm only deals with the vertices within p, which is just a small fraction of the number of vertices in the input graph. If s and t lie in different partitions, then the algorithm only has to deal with the partitions along the shortest path from $P[s]$ to $P[t]$ in the supergraph. Empirically, there are usually 2 to 4 such partitions, which still represents a very small fraction of the input graph.

3.2 Supergraph

We define the supergraph in greater detail and assume that the graph has been partitioned. The super graph $G^S = (V^S, E^S)$ is created according to the following rules: Every vertex u in G belongs to some partition, denoted by $P[u]$, and each partition of G is represented by one vertex in the supergraph. In other words, there is one vertex in the supergraph for each partition of G. Additionally, we include an edge between two vertices in the supergraph if there is an edge crossing between the corresponding partitions. In other words, for any edge $(u,v) \in E$, if $P[u] \neq P[v]$, then we include between vertices $P[u]$ and $P[v]$ in the supergraph.

Algorithm 1 Static Graph Intra-Partition Querying
Input: Graph G and query $q(s,t)$ $(P[s] = P[t])$

```
// Landmark-based approach.
soln ← ∞
for all border vertices b ∈ B(p) do
    if p.dist[b][s] + p.dist[b][t] < soln then
        soln ← p.dist[b][s] + p.dist[b][t]
return soln
```

Algorithm 2 Static Graph Inter-Partition Querying
Input: Graph G and query $q(s,t)$ $(P[s] \neq P[t])$

```
soln ← 0
while P[s] ≠ P[t] do
    p_next ← next[P[s]]
    // Find border vertex in partition closest to u
    // that leads to the next partition in supergraph.
    min_dist ← ∞
    for all w ∈ B(p) do
        if w has neighbor in p_next then
            if P[u].dist[w][s] < min_dist then
                min_dist ← P[s].dist[w][s]
                bv_s ← w
    soln ← soln + min_dist + 1
    s_new ← −1
    // Find a border vertex in the next partition to
    // enter into.
    for all neighbors w of bv_s do
        if P[s] = p_next then
            s_new ← w
            break
    s ← s_new
soln ← soln + P[t].dist[s][t]
return soln
```

3.3 Preprocessing

The preprocessing phase consists of four subphases: 1) Partitioning G 2) Precomputing distances within each partition, 3) Constructing the supergraph G^S, and 4) Precomputing distances in the supergraph.

3.3.1 Partitioning. The first subphase of the preprocessing is the *partitioning* subphase. The output of this partitioning is an array P of size $|V|$ which identifies the partition for each vertex.

The partitioning of the graph can be done using a standard partitioning scheme, such as METIS [18] and Infomap [12]. METIS is a graph partitioner that takes in a graph G and a natural number k, and partitions G into k similarly-sized partitions. Infomap, on the other hand, is a community detection algorithm. It finds some number of highly-connnected communities in a graph. Thus, it only takes G as input and determines an appropriate number of communities by itself.

3.3.2 Distance Computation. In this phase, we find the distance from every border vertex in a given partition p to every other vertex in p. This is achieved by simply running a BFS from each vertex in $B(p)$, all of which can be done in parallel. To increase accuracy within the partition, we can use APSP. In practice, however, this would be a significant increase in preprocessing execution time.

3.3.3 Supergraph Creation. Here, we simply generate the supergraph using the aforementioned rules. Specifically, for all edges $(u, v) \in E$ such that $P[u] \neq P[v]$, we add an edge $(P[u], P[v])$ into the supergraph.

3.3.4 Supergraph Computation. This phase is simply calling APSP on the supergraph. Note that this computationally tractable and affordable since the size of V^S is relatively small. Specifically, our implementation runs a BFS from each vertex for APSP, as this is straightforward and easily parallelizable.

Note that this preprocessing phase only occurs once. Once the graph is preprocessed, our algorithm can answer any number of queries and handle any number of insertions or deletions on the graph if it is dynamic. Thus, any amount of time taken to preprocess the graph will be amortized out after some number of queries and operations. Further, this overhead is negligible for dynamic graph instances where the algorithm is executed for an extended amount of time.

3.4 Querying

Over time, our algorithm will accept a number of queries $q(s_1, t_1), q(s_2, t_2), \ldots$, where each query $q(s_i, t_i)$ is a request for the shortest path from vertex s_i to vertex t_i. There are two cases here: either s and t are in the same partition (Algorithm 1), or they are not (Algorithm 2).

3.4.1 Same Partition. If s and t are in the same partition, we use the landmarks-based algorithm within this partition. In other words, we try to find the shortest path from s to t through some border vertex in the partition they are both in. This can be achieved by iterating over all border vertices b, and minimizing $p.dist[s][b] + p.dist[b][t]$. Recall that the distance from b to any vertex in the partition has already been precomputed. The minimum is our estimate for the shortest path between s and t.

3.4.2 Different Partitions. If s and t are not in the same partition, we find the fewest number of partitions betwen them and the quickest way through each partition through a greedy procedure. Finding the fewest number of partitions between s and t is simply finding the shortest path in the supergraph between $P[s]$ and $P[t]$. Recall that APSP is run on the supergraph in preprocessing, so we have $O(1)$ access to this shortest path. To find the quickest way through each partition, we first have to find the quickest way to exit $P[s]$. This is simply the closest border vertex to s with an edge into partition $next[P[s]]$. Recall that from preprocessing, every border vertex knows the distance between it and every other vertex within its partition. Once we've chosen the border vertex closest to s, we arbitrarily choose one of its neighbors in the next partition to enter into, say bv_{neigh}. After setting $s_{new} = bv_{neigh}$, we can repeat the same procedure as what was done in $P[s]$. Namely, we can iterate over all of the border vertices of $P[s_{new}]$, look at the ones adjacent to the next partition $next[P[s_{new}]]$, and choose the vertex closest to s_{new}. We repeat this procedure until we reach $P[s_{new}] = P[t]$. Once we reach $P[t]$, we simply traverse the shortest path from s_{new} to t, which is known from preprocessing since s_{new} is a border vertex. Once this is done, we have found a path from s to t.

4 DYNAMIC GRAPH ALGORITHM

4.1 Algorithm Descrption

Our dynamic graph algorithm extends our static graph algorithm. Here, we must update our precomputed distances once an edge has been inserted into the graph. The *distance update* phase updates the precomputed distances in preprocessing.

4.2 Distance Update

Our algorithm must update the distances computed in the distance computation phase of preprocessing. When inserting an edge (u, v) into G, there are two cases: either u and v are in the same partition, or they are not.

4.2.1 Same Partition. If u and v are in the same partition p, we must take each border vertex $b \in B(p)$ and update the distance between b and every other vertex within p. These distances were first computed in the preprocessing phase, however they are now outdated and potentially incorrect with the addition or deletion of a new edge. Formally, given a vertex r where $p.dist[r][v]$ is known for all v in the vertex set of p, we must update these distances given a newly inserted edge (u, v). An algorithm described in Green et. al.'s paper on dynamic betweenness centrality solves this exact problem. We need only apply this algorithm for every $r \in B(p)$. Furthermore, recall that partitions are several orders of magnitude smaller than G, and that border vertices represent a fraction of the vertices in p. This makes distance updates in the "same partition" case run very efficiently.

4.2.2 Different Partitions. If u and v are in different partitions, adding an edge (u, v) reduces to adding the edge $(P[u], P[v])$ to the supergraph G^S. This edge changes the distances computed by running APSP on G^S. This is actually the same scenario as the "same partition" case: we have

Table 2: *List of networks used in our experiments. The last column depicts the number of partitions given by Infomap*

Networks for Experiments				
Graph	Type	$\|V\|$	$\|E\|$	Infomap [12] Partitions
in-2004 [2]	Internet	1.3M	13.5M	53
as-skitter [22]	Internet	1.6M	11.0M	182
com-youtube [22]	Social	1.1M	3.0M	951
amazon0601 [22]	Internet	0.40M	3.4M	3
great-britain [2]	Road	7.7M	8.2M	8

$dist[r][v]$ values for a fixed supervertex r and all vertices $v \in V^S$, and we must update them based on an inserted edge. Thus, we can apply Green's algorithm here as well for all $r \in V^S$. Since the supergraph is small, at most a few hundred supervertices, we can update these APSP distances very efficiently.

5 EMPIRICAL ANALYSIS

In this section, we present the performance of our new distance estimation for both the static and dynamic cases. We will evaluate the performance of these algorithm across multiple inputs using several different partitioning strategies. All the results presented in this paper were executed on a quad Intel Xeon E7-4850v3 system, with a total of 56 physical cores (14 per processor) and 112 threads. The core frequency is at 2.80GHz. The system has a total of 2TB of DDR4 RAM, and each of the four processors has a LLC is 35MB.

5.0.1 Graphs. Our algorithm was tested with five undirected and unweighted real-world graphs taken from the DIMACS 10 Graph Challenge set [2] and from Stanford SNAP [22]. The graphs exhibit a wide range of properties and belong to different application fields: social networks, web crawls, and road networks. These graphs are outlined in Table 2. Recall that Infomap decides on the number of partitions rather than getting it as a user parameter. As such we list the number of partitions in Table 2.

5.0.2 Average Relative Error. Recall that our algorithm estimates the distance for each query. Thus, we provide the average relative error for these estimations. To define the average relative error among all queries on the graph, we must first define the error for an individual query. If $d(s,t)$ is the actual shortest path between vertices s and t for a query $q(s,t)$ and $\hat{d}(s,t)$ is our approximation for this distance, then the **query error** in our approximation is $\frac{|d(s,t)-\hat{d}(s,t)|}{d(s,t)}$. The average relative error is just the average of the query errors over all queries. Formally, this is $\frac{1}{|Q|}\sum_{q(s,t)\in Q}\frac{|d(s,t)-\hat{d}(s,t)|}{d(s,t)}$. For case where the exact path is always computed, the ARE is equal to 0.

5.0.3 Partitioning Scheme. In our experiments we use two well-known partitioning algorithms: METIS and Infomap. METIS, a widely used graph partitioner, allows specifying the number of partitions that the original graph should be divided into. For METIS we choose successive powers of 2 as the number of partitions, starting at 2 parititions and making it upto 32 partitions. Altogether 5 different configurations for METIS. Infomap is a community detection based algorithm that partitions a graph using a differnet optimization criteria than METIS. Unlike METIS, it does not take as input the number of partitions (communities) it needs to find. Rather it decides this number by itself.

5.1 Preprocessing Time

For the case of PPSP, where the number of queries is relatively large and can change over time, the costly phase can be answering queries with a pruned SSSP for each query. As discussed earlier in this paper, this can be expensive. In contrast, our pre-processing method is executed once. For the static graph case, the cost of this phase can be amortized across the number of queries. For the dynamic graph case, the cost of this phase can even be further amortized by the fact the dynamic graph algorithm can be executed for indefinite amount of time. This means that the cost of the pre-processing phase, even if it is high, can be ignored if the dynamic graph algorithm can be executed for a very long amount of time.

5.1.1 Distance Computation Sub-phase. We distinguish between two different approaches for the distance estimation. Given a partition: 1) conduct a SSSP from each border vertex (as discussed in the paper) or 2) perform an APSP on the partition. The later of this approaches will improve the quality of the estimation (ensuring 100% accuracy on intra-partition queries) at the cost of an increased execution time. Our experiments use the first (SSSP from the border vertices) of these approaches. In future work we will conduct a more detailed trade-off between these two approach.

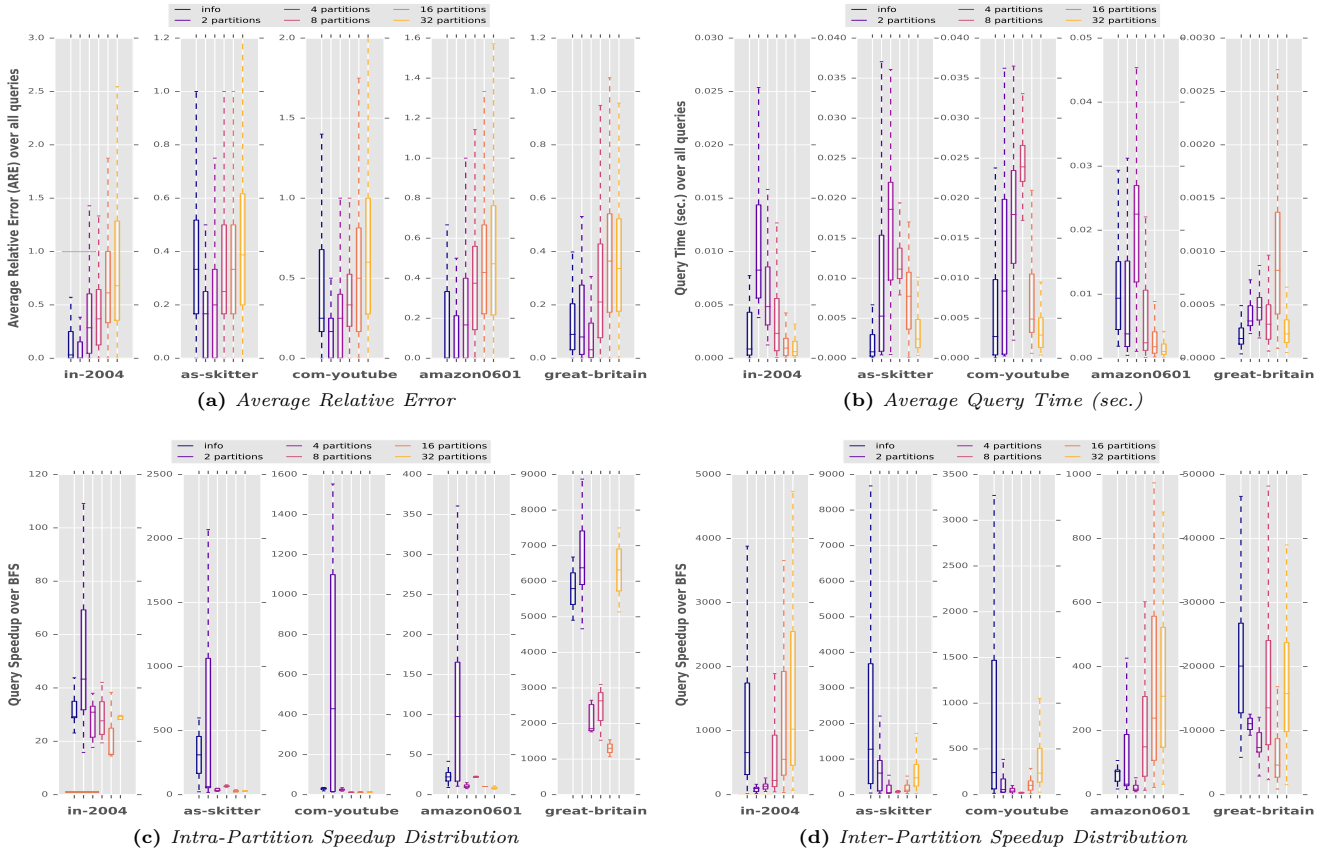
5.1.2 Supergraph Creation and Computation Subphases. The running times of both of these sub-phases relies on the number of partitions generated, which is dependent on the partitioning scheme used. Using a procedure like Infomap generates more partitions and will result in a longer time in this phase. Recall, the supergraph distances uses an APSP computing the distance between the vertices in the supergraph. However, since the number of vertices in the supergraph is typically fewer than a few hundred and the number of edges is a typically an order of magnitude larger, this phase typically does not take a large amount of time despite APSP taking $O((V^S)^3)$ time.

5.2 Static Graph Experiments

There are two main use cases we see for using a static-graph PPSP algorithm, and we run an experiment for each case to assert how well our algorithm performs in each.

The first is relatively straightforward. Given a graph G, the query list Q has arbitrary pairs of vertices in G. For this use case, we measure the time it takes for query $q(s_i,t_i)$ using our algorithm, and compare it with the time for a pruned-BFS starting from s_i and terminating once t_i has been reached.

The second is related to SSSP. Oftentimes, one would like to know the distance between some vertex s and several



(a) *Average Relative Error*

(b) *Average Query Time (sec.)*

(c) *Intra-Partition Speedup Distribution*

(d) *Inter-Partition Speedup Distribution*

fig. 3: *Results for* 100 *random queries experiment*

other vertices in the graph. This can be solved by running SSSP from s. However, running an SSSP can take immense amounts of time, and does an unnecessary amount of work by finding the distance from s to irrelevant vertices in the graph. Another approach to this is to query s with each vertex whose distance is desired. In this experiment, we query the distance from a fixed vertex to several other vertices in the graph and compare the time taken to compute all queries with an SSSP from the fixed vertex.

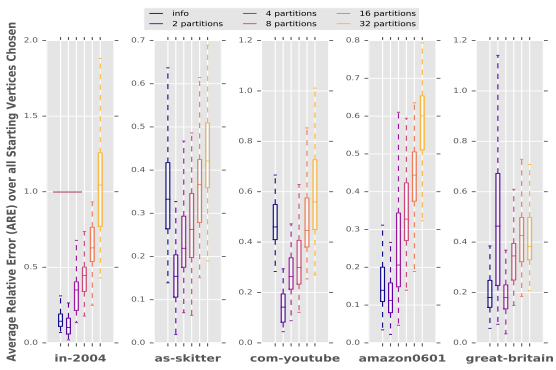
Fig. 3 depicts performance numbers for the random approach where Fig. 4 the performance for the single vertex approach.

Random queries. To evaluate our algorithm for the first of two described scenarios, we run 100 random queries on our static graph algorithm. Each query is a pair of vertices chosen uniformly at random. For each query, we query $q(s, t)$ on our static graph algorithm and measure the accuracy and performance. Then, we run a pruned-BFS, i.e. a BFS from s terminating once t is reached, on the original graph, and compare the performance of that with our static graph algorithm.

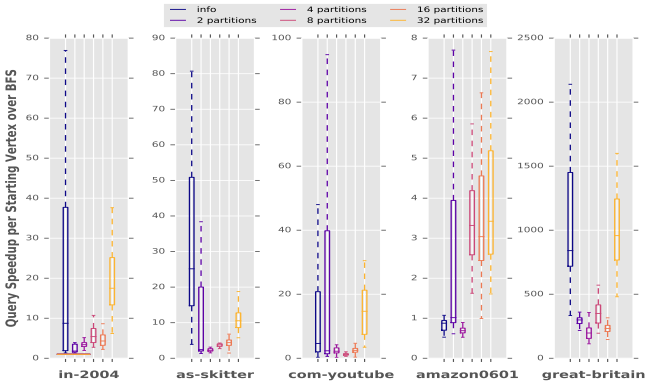
Same initial vertex queries. We also evaluate our algorithm in the second use case, where we have numerous queries with the same start vertex. To do this, we pick some vertex s as the starting vertex uniformly at random. We then choose 100 other vertices to form 100 queries $q(s, v_1), \ldots, q(s, v_{100})$, and run our static graph algorithm on each of these queries. Then, we run a full BFS from s on the whole graph, and compare the time taken for all 100 queries to the time for the whole BFS. For completeness, we run this experiment on 50 different start vertices.

Performance Trends. First of all we note that the ARE tends to increase with the number of partitions. An explanation for this is, on average, the intra-partition query cases tend to be more accurate than the inter-partition query cases. Thus, with more partitions, inter-partition cases occur more often than intra-partition cases.

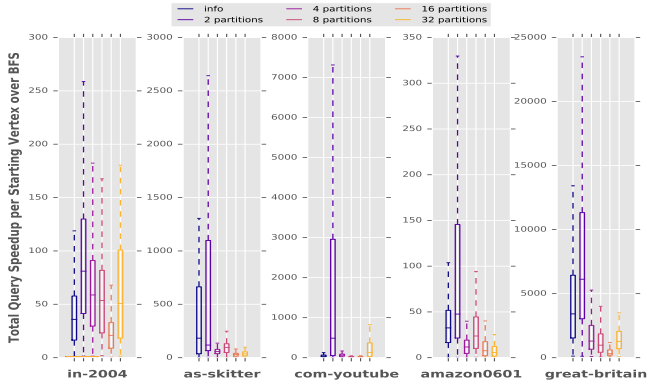
The AQT (average query times) has a slightly different trend. Initially, as the number of partitions increases so does the AQT. However, at some point, when the number of partitions increases, the query time decreases. With more partitions, inter-partition queries occur more often. Also, there are fewer border vertices that need to be queried by the APSP in the supergraph. This reduces the amount of



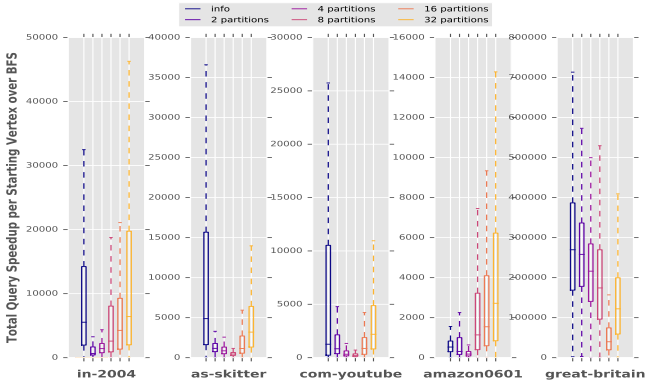
(a) *Average Relative Error*



(b) *Average Query Speedup over BFS*



(c) *Intra-Partition Speedup Distribution over pruned-BFS*



(d) *Inter-Partition Speedup Distribution over pruned-BFS*

fig. 4: *Results for same initial vertex experiment distributed across all initial vertices*

time taken by both intra-partition queries and inter-partition queries. The combination of these observations results in the decreased AQT with more partitions, albeit with some variance.

Overall, the ARE increases with more partitions while the AQT decreases with more partitions. Thus, one can view the choice of partition count as a tradeoff between accuracy and efficiency. In applications where accuracy is more critical than efficiency, fewer partitions should be used. On the other hand, in applications where accuracy is not as important as efficient queries, more partitions should be used.

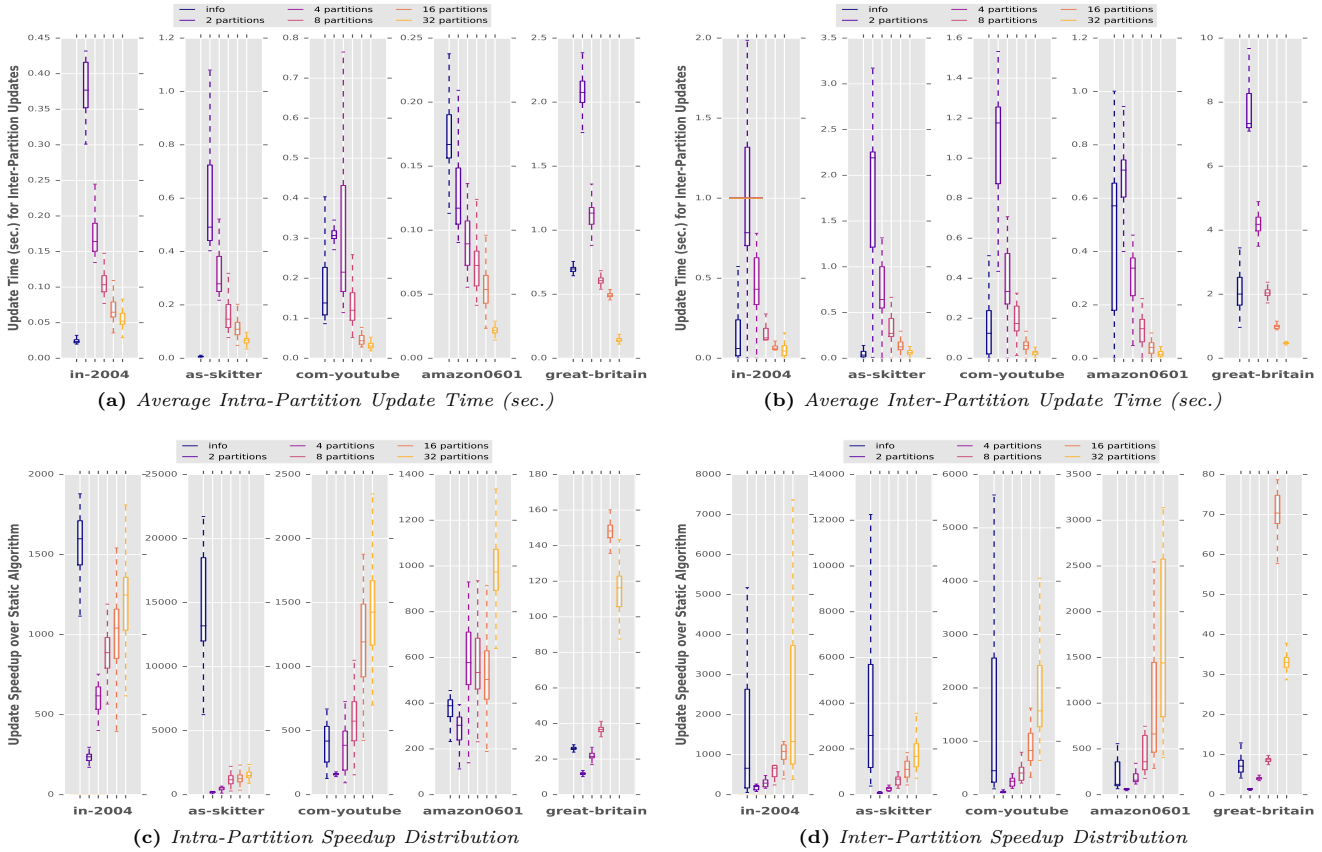
Partitioner comparison. For both test cases, Fig. 3 (a) and Fig. 4 (a), we can see the difference in the quality of the estimate of using METIS (with a different number of partitions) against Infomap. Note, that in three out of five cases, Infomap creates a large number of partitions. From a qualitative perspective, the partitions given by Infomap enable estimating distances in a manner that is equal to the ones given by METIS for a small number of partitions. From a performance perspective, using the (b) subplots, it seems that the Infomap subgraph also allows for faster queries in comparison to its METIS counterparts.

5.3 Dynamic Graph Experiments

For the case of dynamic graphs we focus on the performance of our dynamic distance updating mechanism. While our algorithm works for both edge insertions and deletions, for simplicity we primarily focus on edge insertions. Note, our new algorithm is deterministic such that if $G' = G \cup \{(u, v)\}$, then any query on G after inserting edge (u, v) would return the same result as if G' were the original input graph. As such a qualitative analysis on the distance estimation is not necessary.

We compare the time it takes to update the distances in the partition graph and supergraph with the time it takes to compute the initial distances in the preprocessing phase of the static graph. Note, such an analysis does not capture the full speedup of our algorithm over running the pruned BFS algorithm that was used for the analysis of the static graph algorithm. Such a comparison would lead to even higher speedups of our algorithm over the static case.

To evaluate our dynamic graph algorithm, we differentiate between intra-partition insertions and inter-partition insertions. For **intra-partition** insertions, we pick an arbitrary partition $p \in P$, pick 100 random edges in p, and remove them from p. We then reinsert each edge into the graph and



(a) *Average Intra-Partition Update Time (sec.)*

(b) *Average Inter-Partition Update Time (sec.)*

(c) *Intra-Partition Speedup Distribution*

(d) *Inter-Partition Speedup Distribution*

fig. 5: *Results for dynamic graph algorithm experiments*

measure the amount of time taken to update distances in p for each edge. We ensure that p does not become disconnected after removing the 100 edges. A similar experiment is run for **inter-partitions** insertions. We pick 100 arbitrary edges in G. Let (u_i, v_i) be one such edge. We then remove edge (u_i, v_i) from G and edge $(P[u_i], P[v_i])$ from the supergraph. We then time how long it takes to update distances and shortest paths in the supergraph for each edge after reinserting each edge. After computing the time taken to run the dynamic graph algorithm on each update, we compare each time with the amount of time taken to compute all distances in the preprocessing. This gives us the amount of speedup our dynamic graph algorithm has over rerunning the preprocessing our static graph algorithm. Fig. 5 depicts the performance of our dynamic graph algorithm and shows both raw execution times as well as the speedups.

One noticeable trend is that the execution time of our dynamic algorithm seems to improve with the number of partitions as the average size of the partitions decrease [1]. Theoretically, this trend should reverse with a sufficient number of partitions, as the extreme would be having $|V|$ partitions. This would reduce to the computationally expensive APSP problem. It is relatively straightforward to see why intra-partition updates are more efficient with more partitions.

Counterintuitively, inter-partition updates are also more efficient with a larger number of partitions. When an edge (u, v) is inserted between partitions, there is a chance that u or v becomes a new border vertex in their respective partitions. If u is a new border vertex in $P[u]$, then we must run a BFS from u within partition $P[u]$. The same applies to vertex v within $P[v]$. However, we will not have to perform as many of these BFS's as additional partitions are used. The reason for this is that with more partitions, each vertex has a greater chance of having an edge to a vertex in another partition.

In other words, with more partitions, the proportion of border vertices in each partition increases. This means that when adding an edge (u, v), there is a smaller chance that u or v is a new border vertex. Since these BFS's do not occur as often with a larger number of partitions, more partitions makes inter-partition updates run more efficiently. Since the number of partitions here is kept small when compared to the vertex set of the graph, the actual inter-partition distance update phase of the dynamic graph algorithm is not as

[1] A large number of partitions implies that each partition is also smaller, and, since each intra-partition update only works on a single partition a smaller number partition size would also lead to less work.

expensive as running all of these BFS's. As mentioned above, this would not be the case with a substantially larger number of partitions. However, having a significantly larger number of partitions would not be useful in practice since the ARE would grow immensely large, as illustrated with the static graph algorithm.

This further emphasizes how the choice of partitions is a tradeoff between accuracy and efficiency, as was noted for the static graph algorithm. For the dynamic graph algorithm, updates are more efficient with more partitions. Yet, as we highlighted for the static case, increasing the number of partitions can hurt the quality of the estimation.

6 CONCLUSIONS

We have shown algorithms that run efficiently on both static and dynamic graphs. On static graphs, our first algorithm is as efficient as those already in the literature. However, none of those algorithms, to our knowledge, are meant to be run on dynamic graphs. If one were to modify existing algorithms to run on dynamic graphs, they would be extremely computationally expensive on edge insertions and deletions. However, we extend our static graph algorithm to handle dynamic graphs. Our second algorithm is orders of magnitude faster on dynamic graphs than other algorithms in the literature. At its core, the reason our algorithm is so efficient on dynamic graphs is that it only needs to focus on a small fraction of the graph (i.e. a partition) on each edge insertion or deletion, whereas other algorithms must operate on the whole input graph.

For future work, we will investigate ways to intelligently determine the optimal number of partitions to choose a priori. Additionally, interest in distributed algorithms has been increasing recently due to the growth of graph size outpacing the growth of single-machine-RAM size. One avenue for this is by partitioning an input graph and assigning each partition to a seperate node or accelerator. Our algorithm would fit well in this model, since most edge updates result in computations within an individual partition.

REFERENCES

[1] D. Bader, J. Berry, A. Amos-Binks, D. Chavarra-Miranda, C. Hastings, K. Madduri, and S. C. Poulos. STINGER: Spatio-Temporal Interaction Networks and Graphs (STING) Extensible Representation. Technical report, Georgia Institute of Technology, 2009.

[2] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, editors. *Graph Partitioning and Graph Clustering. 10th DIMACS Implementation Challenge Workshop*, number 588 in Contemporary Mathematics, 2013.

[3] E. Bergamini, H. Meyerhenke, and C. L. Staudt. Approximating betweenness centrality in large evolving networks. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 133–146. SIAM, 2015.

[4] P. Berman and S. P. Kasiviswanathan. Faster approximation of distances in graphs. In *Proceedings of the 10th international conference on Algorithms and Data Structures*, pages 541–552. ACM, 2007.

[5] F. Busato, O. Green, N. Bombieri, and D. Bader. Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices on GPUs. In *IEEE Proc. High Performance Extreme Computing (HPEC)*, Waltham, MA, 2018.

[6] G. F. I. C. Demetrescu. Experimental analysis of dynamic all pairs shortest path algorithms. *ACM Transactions on Algorithms (TALG)*, 2:578–601, 2006.

[7] G. F. I. C. Demetrescu. Fully dynamic all pairs shortest paths with real edge weights. *Journal of Computer and System Sciences*, 72:813–837, 2006.

[8] S. Chechik, D. H. Larkin, L. Roditty, G. Schoenebeck, R. E. Tarjan, and V. V. Williams. Better approximation algorithms for the graph diameter. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 1041–1052. ACM, 2014.

[9] P. Crescenzi, R. Grossi, L. Lanzi, and A. Marino. A comparison of three algorithms for approximating the distance distribution in real-world graphs. In *Proceedings of the First international ICST conference on Theory and practice of algorithms in (computer) systems*, pages 92–103. ICST, 2011.

[10] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: A Framework for Mapping Complex Scientific Workflows onto Distributed Systems. *Scientific Programming*, 13(3):219–237, 2005.

[11] D. Ediger, R. McColl, J. Reidy, and D. Bader. STINGER: High Performance Data Structure for Streaming Graphs. In *IEEE High Performance Embedded Computing Workshop (HPEC 2012)*, pages 1–5. IEEE, 2012.

[12] D. Edler and M. Rosvall. The MapEquation software package.

[13] A. V. Goldberg. Point-to-Point Shortest Path with Preprocessing. In *SOFSEM '07 Proceedings of the 33rd conference on Current Trends in Theory and Practice of Computer Science*, pages 88–102. Austrian Computer Society, 2007.

[14] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for a*: Efficient point-to-point shortest path algorithms. Technical report, MSR-TR-2005-132, 2005.

[15] O. Green and D. Bader. cuSTINGER: Supporting Dynamic Graph Algorithms for GPUS. In *IEEE Proc. High Performance Extreme Computing (HPEC)*, Waltham, MA, 2016.

[16] O. Green, R. McColl, and D. Bader. A Fast Algorithm For Streaming Betweenness Centrality. In *4th ASE/IEEE International Conference on Social Computing (SocialCom)*, 2012.

[17] M. Han and K. Daudjee. Giraph unchained: barrierless asynchronous parallel execution in pregel-like graph processing systems. In *Proceedings of the VLDB Endowment*, pages 950–961. VLDB Endowment, 2015.

[18] G. Karypis and V. Kumar. Metis-unstructured graph partitioning and sparse matrix ordering system, version 2.0. 1995.

[19] M. Kas, M. Wachs, K. M. Carley, and L. R. Carley. Incremental algorithm for updating betweenness centrality in dynamically growing networks. In *Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pages 33–40. IEEE/ACM, 2013.

[20] V. King. Fully Dynamic Algorithms for Maintaining All-Pairs Shortest Paths and Transitive Closure in Digraphs. In *Proceeding FOCS '99 Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 81. ACM, 1999.

[21] M.-J. Lee, J. Lee, J. Y. Park, R. H. Choi, and C.-W. Chung. QUBE: a Quick algorithm for Updating BEtweenness centrality. In *ACM Int'l Conf. on World Wide Web*, pages 351–360, 2012.

[22] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June.

[23] Y. Low, D. B. J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. In *Proceedings of the VLDB Endowment*, pages 716–727. VLDB Endowment, 2012.

[24] M. Nasre, M. Pontecorvi, and V. Ramachandran. Betweenness centrality - incremental and faster. In *Mathematical Foundations of Computer Science 2014 - 39th International Symposium, MFCS 2014, Budapest, Hungary, August 25-29, 2014. Proceedings, Part II*, pages 577–588, 2014.

[25] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis. Fast shortest path distance estimation in large networks. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 867–876. ACM, 2009.

[26] G. Ramalingam and T. Reps. An Incremental Algorithm for a Generalization of the Shortest-Path Problem. *Journal of Algorithms*, 21:267–305, 1996.

[27] J. Shun. An Evaluation of Parallel Eccentricity Estimation Algorithms on Undirected Real-World Graphs. In *Proceedings of the ACM Conference on Knowledge Discovery and Data Mining (KDD)*, pages 1095–1104. ACM, 2015.